Chapter 3

The Signal Processing Style of Programming

The *signal processing style (SPS)* of programming is a label for the class of techniques that organize programs like the sliver diagrams introduced in the previous chapter. In this style, computations are are expressed as networks of computational devices that generate, map, filter, and accumulate data transmitted over directed cables. This style encompasses a wide range of programming techniques used in functional, imperative, object oriented, and concurrent languages. The name "signal processing style" is suggested by the resemblance between sliver diagrams and signal processing block diagrams.

This chapter explores the tension between modularity and control in the signal processing style of programming. It elucidates two key points about this style:

- The signal processing style is a powerful means of decomposing programs into modular units that encapsulate important programming idioms.
- Classical techniques for programming in the signal processing style often preempt the programmer from controlling important operational aspects of programs expressed in this style (e.g., execution time, space complexity, operation scheduling).

The remainder of this dissertation explores ways of reducing the tension between modularity and control. In particular, the lock step processing model developed later is able to achieve modularity while preserving space complexity and operation scheduling (it does not address the control of execution time).

I will classify SPS techniques into two categories according to how they represent the slivers and cables in a sliver diagram:

1. The *aggregate data approach* represent slivers as operators that manipulate aggregate data (e.g., lists, streams, arrays, trees) and cables as the aggregate data structures themselves.

2. The *channel approach* represent slivers as communicating processes and cables as the communication channels between processes.

The lines between these approaches are not rigid; we will encounter techiques that exhibit characteristics of both.

In the following sections, we study these two approaches in the context of the database and alpha renaming examples. We will also consider several other techniques that resist classification into the above two categories. We will see that each of the standard SPS techniques is a two-edged sword: it helps the programmer subdue complexity, but it also either (1) prevents the programmer from controlling important operational behavior or (2) unduly limits the class of programs that can be expressed.

3.1 The Aggregate Data Approach

A common technique for encoding sliver diagrams is to represent slivers as procedures that manipulate aggregate data structures. Then the slivers can be wired together simply by the usual methods for procedural composition. For example, the following procedures are the textual encodings of the sliver diagrams for `fat-cats` (Figure 2.9) and `mean-age` (Figure 2.11):

```
(define (fat-cats database)
  (stacking-collect-list
   (map-name
    (filter-salary threshold
                   (generate-records database)))))

(define (mean-age database)
  (let ((records (generate-records database)))
    (/ (running-sum (map-age records))
       (running-sum (map-one records)))))
```

Here we assume that each procedure called in the body corresponds to the similarly-named sliver. Note how the wire connections are represented by the natural data dependences of nested subexpressions; in some sense, the aggregate values *are* the cables. Cable fan-out is handled by `let`, whose bindings allow the same aggregate value to be used by more than one procedure.

The aggregate data style has its roots in Lisp's list manipulation routines (as epitomized by `mapcar`) and APL's array operators. Today, the aggregate data approach is the main organizing principle for data parallel languages (e.g., Fortran 90 [Ame89], C* [RS87], NESL [Ble92], paralations [Sab88]). It is also a commonly used technique in many other languages, especially functional and mostly functional ones (e.g., Haskell [HJW$^+$92], Id [AN89], Common Lisp [Ste90], Scheme [CR$^+$91], ML [MTH90]).

3.1.1 Database Example: A List Implementation

For the linear database example, lists are an obvious choice for the type of aggregate data structure. Figure 3.1 shows how each of the slivers can be represented as a list-manipulation procedure.

Expressing `mean-age` and `fat-cats` as combinations of parts that share a standard interface (lists) is a powerful strategy because the parts are reusable in a mix and match way. Even though the code size for the aggregate data versions of these two programs (including the definitions of their subroutines) is much larger than the size of the monolithic versions, this increased size is offset by the modularity of the structure and the fact that the size of the subroutines should be be amortized over all the places where those subroutines will be used. Parts like those in Figure 3.1 are likely to be used pervasively in database applications. And parts like `map-one`$_{list}$ and `running-sum`$_{list}$ have even broader applicability, since they

```
;;; Generators
(define (generate-records_list database)
  (define (gen record)
    (if (end-of-database? record)
        '()
        (cons record (gen (next-record record)))))
  (gen (first-record database)))

;;; Mappers
(define (map-age_list records)
  (if (null? records)
      '()
      (cons (record-get (car records) 'age)
            (map-age_list (cdr records)))))

(define (map-one_list list)
  (if (null? list)
      '()
      (cons 1 (map-one_list (cdr list)))))

(define (map-name_list records)
  (if (null? records)
      '()
      (cons (record-get (car records) 'name)
            (map-age_list (cdr records)))))

;;; Filters
(define (filter-salary_database threshold records)
  (cond ((null? records) '())
        ((> (record-get (car records) 'salary) threshold)
         (cons (car records)
               (filter-salary_database threshold (cdr records))))
        (else (filter-salary_database threshold (cdr records)))))

;;; Accumulators
(define (running-sum_list list)
  (define (accum lst sum)
    (if (null? lst)
        sum
        (accum (cdr lst) (+ (car lst) sum))))
  (accum list 0))

(define (stacking-collect-list_list lst) lst) ; Already a list!
```

Figure 3.1: List-based implementation of the slivers used by `mean-age` and `fat-cats`.

will work on any linear structures, not only databases. This underscores the key advantage of the aggregate data approach: aggregate data operators constitute a *language* for working in a domain.

Using higher-order procedures, it is possible to obtain modular pieces boasting even greater reusability. Figure 3.2 introduces higher-order procedures that capture the essence of generation, mapping, and iterative accumulation. The `mean-age` subroutines are just instances of these more general abstractions (Figure 3.3).

```
(define (generate_list done? next current)
  (if (done? current)
      '()
      (cons current (generate_list done? next (next current)))))

(define (map_list function lst)
  (if (null? lst)
      '()
      (cons (function (car lst))
            (map_list function (cdr lst)))))

(define (filter_list predicate lst)
  (cond ((null? lst) '())
        ((predicate (car lst))
         (cons (car lst) (filter_list predicate (cdr lst))))
        (else (filter_list predicate (cdr lst)))))

(define (iter-accumulate_list operator identity list)
  (define (acuum lst ans)
    (if (null? lst)
        ans
        (accum (cdr lst) (operator (car lst) ans))))
  (accum list identity))
```

Figure 3.2: Higher-order list-manipulation procedures.

It is worth noting how the list-based implementation finesses the modularity problems with filtering for the sliver decomposition of Figure 2.7. A list representing a cable does not explicitly represent any presence flags. Rather, any element whose associated presence flag is false is simply not included in the list. This *compression* technique enhances modularity because there is no need to have different procedures for handling filtered vs. unfiltered data. Unfortunately, the compression trick does not extend to the tree-structured case; removing an element from a list leaves a list, but removing an element from a tree does not

```
(define (generate-records_list database)
  (generate/list end-of-database? next-record (first-record database)))

(define (map-age_list records)
  (map/list (lambda (rec) (record-get rec 'age)) records))

(define (map-one_list list)
  (map/list (lambda (rec) 1) list))

(define (running-sum_list list)
  (iter-accumulate/list + 0 list))
```

Figure 3.3: Implementations of the `mean-age` subroutines in terms of the higher-order list-manipulation procedures.

leave a tree! Later, we will see how to deal with this problem.

As an aside, it should be mentioned that the list-based programs are really only simulating the sliver diagrams for `mean-age` and `fat-cats`. Indeed, each of the procedures in Figure 3.1 can be described by its own sliver diagram! For example, the structure of `fat-cats`$_{list}$ is really as shown in Figure 3.4. Here, each of the four component procedures is associated with a dotted box containing the sliver diagram for that procedure. The core slivers (`GENERATE-RECORDS`, `FILTER-SALARY`, `MAP-NAME`) are wrapped in instances of `SPLAY-LIST` and `GLOM-LIST` that convert between the standard external list representation and the internal cable representation: `SPLAY-LIST` is a generator that, given a list, spits out its elements in order to a cable; `GLOM-LIST` is just a synonynm for `STACKING-LIST-COLLECT`. (`SPLAY-LIST` and `GLOM-LIST` are the analogues of buffers in electrical systems that present standard impedances to the rest of a circuit.) Each dotted box simulates the input/output behavior of the corresponding sliver in Figure 2.7, while each wire between dotted boxes carries a list that encodes an entire cable structure between slivers.

It is possible to imagine doing algebra on slivers. For example, `SPLAY-LIST` and `GLOM-LIST` act as inverses, so it should be possible to treat any juxtaposition of these two slivers as an identity. Using this fact in conjunction with the equivalence of `GLOM-LIST` and `STACKING-LIST-COLLECT`, it is easy to show that the diagram in Figure 3.4 is "equivalent" to the diagram in Figure 2.9. But this notion of equivalence only captures the input/output behavior of `fat-cats`. It does not capture how `fat-cats`$_{list}$ builds and takes

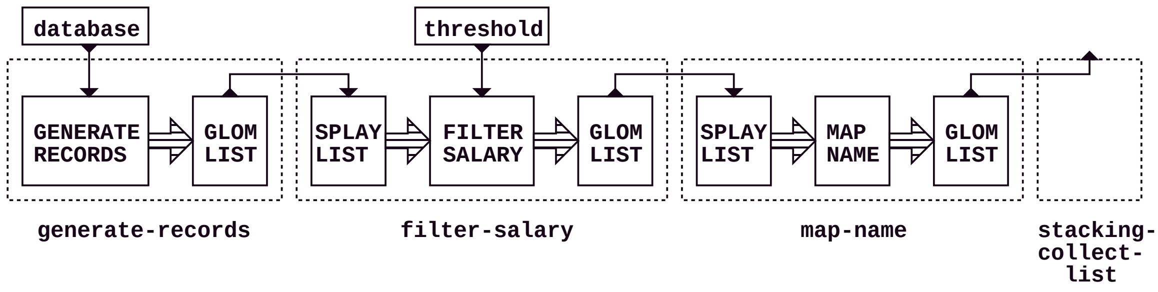


Figure 3.4: Sliver diagram for the list-based implementation of `fat-cats`. Each of the dotted boxes corresponds to the procedure whose name appears below it.

aparts intermediate lists that do not exist in `fat-cats`$_{fun}$. Nor does it capture how (in Scheme) `fat-cats`$_{list}$ performs all generation steps before any filtering steps (in contrast, `fat-cats`$_{fun}$ interleaves the steps from these stages).

3.1.2 Database Example: An Array Implementation

Lists are not the only data structure that can be used to implement the database examples in the aggregate data style. In a data-parallel language, the natural representation for cables would be some sort of array. In such a language, the basic strategy for the database examples would be to fill an array with all the database records and then use appropriate data-parallel array operators to derive a result.

A notable feature of data-parallel operators is that they tend to operate in an elementwise fashion on the input array. In this paradigm, a natural way to do filtering is to first compute a boolean array containing presence flags for the corresponding elements of an input array, and later use the presence flags to select the array indices at which an operation will be performed. We will call this approach the *gap* technique of filtering, because it models the absence of an element with an explicit marker indicating that the element is not there. Via the selection mechanism, all array operators are equipped to handled filtered data at all times; unfiltered data is just the special case where all indices are selected. Thus, modularity is enhanced in the gap technique by having every sliver handle filtered data.

3.1.3 Alpha Renaming Example: A Tree Implementation

Significant gains in modularity can be achieved by expressing alpha renaming as an SPS program. The sliver diagram in Figure 2.15 is easy to express in the aggregate data approach:

```
(define (alpha-rename_tree exp)
  (let ((exp-tree (splay-tree exp)))
    (let ((def-tree (filter-formals exp-tree)))
      (glom-tree (rename exp-tree
                         (environment def-tree
                                      (fresh-names def-tree)))))))
```

Since each of the conceptual processes involved in alpha renaming is some sort of tree walk, it is natural to use trees as the common currency through which the slivers communicate. To simplify the implementation, all of the intermediate trees have a common form. They are instances of a tree data abstraction in which every tree node is an immutable structure maintaining a single value (the *datum*) and a list of subtrees. The `tree` constructor takes a datum and list of subtrees and returns a newly constructed tree node with this information; the `tree-datum` and `tree-subtrees` selectors extract information from a tree. Here is a straightforward implementation of this abstraction that we will assume in our examples.

```
(define (tree value subtrees) (cons value subtrees))

(define (tree-datum tr) (car tr))

(define (tree-subtrees tr) (cdr tr))
```

The datum stored at a tree node can be any value. Multiple values can be stored at a tree node by packaging them up into a single compound datum. A tree whose subtree list is empty is called a *leaf.*

Figures 3.5 – 3.7 present the tree-based implementations of the subroutines used by `alpha-rename`$_{tree}$. Although lambda calculus terms have an inherent tree structure, it is still necessary to convert them into the common tree format used by the other operations. We will refer to the original syntactic form as an *term* and the converted one as a *node tree*. The conversion between terms and node trees is handled by `splay-tree` and `glom-tree`, as shown in Figure 3.5. The datum of an node tree, called a *node*, is a list of a term type and

```
(define (splay-tree exp)
  (cond
   ((variable? exp)
    (tree (list 'variable exp) '()))
   ((lambda? exp)
    (tree (list 'lambda (formal exp))
          (list (splay->tree (body exp)))))
   ((call? exp)
    (tree (list 'call)
          (list (splay->tree (rator exp))
                (splay->tree (rand exp)))))
   ))

(define (glom-tree tr)
  (let ((exp (tree-datum tr))
        (subexps (tree-subtrees tr)))
    (case (first exp)
     ((variable) (second exp))
     ((lambda)
      (make-lambda (second exp)
                   (glom-tree (first subexps))))

     ((call)
      (make-call (glom-tree (first subexps))
                 (glom-tree (second subexps))))
     )))

(define (filter-formals exp-tree)
  (let ((exp (tree-datum exp-tree))
        (subexps (tree-subtrees exp-tree)))
    (if (eq? 'lambda (first exp))
        (tree (second exp)
              (list (filter-formals (first subexps))))
        (tree '() ; No names declared at this node.
              (map filter-formals subexps)))))
```

Figure 3.5: Subroutines for the alpha-renamer, part I.

extra information relevant to that type. The subtrees of an node tree are just the converted subterms.

For instance, consider the following sample term:

```
(define a-term
  '(lambda x
     (lambda y
       (call (lambda x
               x)
             (lambda y
               x)))))
```

The node tree corresponding to `a-term` is given below:

```
(splay-tree a-term)
```
$\stackrel{\text{eval}}{\Longrightarrow}$
```
((lambda x)
 ((lambda y)
  ((call)
   ((lambda x)
    ((variable x)))
   ((lambda y)
    ((variable x))))))
```

The `filter-formals` procedure (see Figure 3.5) highlights the locations of the declared names in an expression tree. It returns a *name tree* with the same shape as its input in which every `lambda` node has been mapped to its formal parameter and every non-lambda node has been mapped to nil. This is an example of the gap technique of filtering applied to trees; here, a gap is explicitly represented by a nil. For example,

```
(filter-formals (splay-tree a-term))
```
$\stackrel{\text{eval}}{\Longrightarrow}$
```
(x
 (y
  (()
   (x
    (()))
   (y
    (()))))))
```

The name generator, `fresh-names` (see Figure 3.6), is the most complicated component in this decomposition. It transforms a given name tree into a new name tree in which all the names are distinct. It does this in two stages. First, it performs a left-to-right preorder walk over the given name tree, in which it increments a counter every time it encounters a name. This process returns a *number tree* with the same shape as the name tree in which every node is decorated with the value of the counter from the walk. Second, it maps a new-symbol generator over the name tree and the number tree to get a tree of fresh names. For example:

```
(lr-pre-number (filter-formals (splay-tree a-term) 0))
```
$\stackrel{\text{eval}}{\Longrightarrow}$
```
(0
 (1
  (2
   (2
    (3))
   (3
    (4)))))
```

```
(fresh-names (filter-formals (splay-tree test)))
```
$\stackrel{\text{eval}}{\Longrightarrow}$
```
(x_0
 (y_1
  (()
   (x_2
    (()))
   (y_3
    (())))))
```

It is worth noting that `lr-pre-number` works on any tree, not only name trees.

The `environment` procedure (Figure 3.7) takes a name tree and a value tree and creates a tree of environments that has the same shape as the two input trees. The environment datum at every node in the resulting tree is the environment of the parent node extended by a binding between the corresponding name and value. When there is no corresponding name (i.e., the name tree has a nil at a node), the environment is passed down unchanged from above. If we assume that environments are implemented as association lists (where earlier bindings take precedence over later ones), then `environment` has the following behavior on our test expression:

```
(let ((defs (filter-formals (splay-tree a-term))))
  (environment defs
               (fresh-names defs)))
```
$\stackrel{\text{eval}}{\Longrightarrow}$
```
(((x . x_0))
 (((y . y_1) (x . x_0))
  (((y . y_1) (x . x_0))
   (((x . x_2) (y . y_1) (x . x_0))
    (((x . x_2) (y . y_1) (x . x_0))))
   (((y . y_3) (y . y_1) (x . x_0))
    (((y . y_3) (y . y_1) (x . x_0)))))))
```

```
(define (fresh-names name-tree)
  (map-fresh name-tree
             (lr-pre-number name-tree 0)))

(define (lr-pre-number tr num)
  ;; Create a number tree for the given tree
  ;; by a left-to-right preorder walk.
  (define (walk tr num)
    ;; Returns a list of a new tree and a new num
    (mlet (((num-subtrees num1)
            (walk-trees (tree-subtrees tr)
                        (if (null? (tree-datum tr))
                            num
                            (+ num 1)))))
      (list (tree num num-subtrees)
            num1)))
  (define (walk-trees trs num)
    ;; Returns a list of a new tree list and a new num
    (if (null? trs)
        (list '() num)
        (mlet (((first-tree num1) (walk (car trs) num)))
          (mlet (((rest-trees num2) (walk-trees (cdr trs) num1)))
            (list (cons first-tree rest-trees)
                  num2)))))
  (car (walk tr 0)))

(define (map-fresh name-tree num-tree)
  (let ((name (tree-datum name-tree))
        (num  (tree-datum num-tree)))
    (tree (if (null? name)
              '()
              (fresh-name name num))
          (map map-fresh
               (tree-subtrees name-tree)
               (tree-subtrees num-tree)))))
```

Figure 3.6: Subroutines for the alpha-renamer, part II.

```
(define (environment name-tree val-tree)
  ;; Assume ENV-IDENTITY binds each name to itself
  ((env-down env-identity) name-tree val-tree))

(define (env-down env)
  (lambda (ntree vtree)
    (let ((name (tree-datum ntree))
          (val  (tree-datum vtree)))
      (let ((new-env (if (null? name)
                         env
                         (env-extend name val env))))
        (tree new-env
              (map (down new-env)
                   (tree-subtrees ntree)
                   (tree-subtrees vtree)
                   ))))))

(define (rename exp-tree env-tree)
  (let ((exp (tree-datum exp-tree))
        (env (tree-datum env-tree)))
    (tree (case (first exp)
            ((variable)
             (list 'variable (env-lookup (second exp) env)))
            ((lambda)
             (list 'lambda (env-lookup (second exp) env)))
            (else exp))
          (map rename
               (tree-subtrees exp-tree)
               (tree-subtrees env-tree)))))
```

Figure 3.7: Subroutines for the alpha-renamer, part III.

The final component of the decomposition is the `rename` procedure (Figure 3.7). It takes a node tree and an environment tree and returns a new expression tree in which all lambda-bound variables and variable references are renamed according to the corresponding environment. In the case of our running example:

```
(let ((exp-tree (splay-tree a-term)))
  (let ((defs (filter-formals exp-tree)))
    (rename exp-tree
            (environment defs
                         (fresh-names defs)))))
```

$\overset{\text{eval}}{\Longrightarrow}$

```
((lambda x_0)
 ((lambda y_1)
  ((call)
   ((lambda x_2)
    ((variable x_2)))
   ((lambda y_3)
    ((variable x_0))))))
```

Putting all the components together yields the complete alpha renamer, which works as advertised on the test term:

```
(alpha-rename a-term)
```

$\overset{\text{eval}}{\Longrightarrow}$

```
(lambda x_0
  (lambda y_1
    (call
     (lambda x_2
       x_2)
     (lambda y_3
       x_0))))
```

The signal processing organization for the alpha renaming program has numerous modularity advantages over the monolithic approaches:

- The tree-manipulation procedures can be designed, implemented, and debugged independently. In the monolithic versions, it was impossible to perform any of these tasks on only one of the subprocesses since they all were intertwined.

- Many program modifications can be made by local changes to the tree-manipulation procedures. For example, in order to make a name generator that walks right-to-left rather than left-to-right, it is only necessary to replace the lr-pre-number within `fresh-names` by an appropriately defined `rl-pre-number`. Such a modification is entirely local; no other module need be changed.

- The tree-manipulation procedures are reusable. Structurally, they share a standard interface (trees) that makes them easy to mix and match. More importantly, the modules can be designed to make minimal assumptions about the contexts in which they are to be used, thereby broadening their range of applicability. For example, the `environment` procedure embodies a downward flow of binding information for *any* tree of names and *any* tree of values, as long as they have the same shape. More context-dependent modules, such as `filter-formals` and `rename` (both of which contain references to lambda-calculus specific details like the `lambda` keyword) can be generalized to make them more widely applicable.

- Many of the tree-manipulation procedures share a common structure that can be captured by higher-order tree-manipulation procedures. For example, `filter-formals` and `rename` are both instances of a more general tree-mapping process in which each datum of a result node only depends on data of the argument nodes. `Environment`, `glom-tree`, and `lr-pre-number` are instances, respectively, of more general processes that accumulate information down a tree, up a tree, and threaded through a tree. We will consider such generalizations in depth in Chapter 6.

- The modules can capture default behavior in a way that facilitates extensions. Extending the alpha renamer to deal with an `if` construct only requires adding new handlers to the conversion routines `splay-tree` and `glom-tree`:

```
(define (splay-tree exp)
  (cond
   ⋮
   ((if? exp)
    (tree (list 'if)
          (list (splay-tree (test exp))
                (splay-tree (then exp))
                (splay-tree (else exp)))))
   ⋮
   ))
```

```
(define (glom-tree tr)
  (let ((exp (tree-datum tr))
        (subexps (tree-subtrees tr)))
    (case (first exp)
     ⋮
      ((if) (make-if (glom-tree (first subexps))
                     (glom-tree (second subexps))
                     (glom-tree (third subexps))))
      ⋮
      )))
```

In particular, the core modules of the alpha renamer — name generation, environment, renaming — need not be changed at all to handle the `if` construct! The reason is that these modules all have appropriate defaults built in. For example, at any node where a name is not supplied, `environment` simply passes the current environment down to all the subexpressions. This default captures essential behavior of "environmentness" that need be specified in only one place.

To be fair, it is worth noting that adding a new name declaration construct would not be as easy as adding `if`. The problem is that the notions of name declaration and scope are too closely tied to details of the lambda calculus within `filter-formals` and `rename`. However it is possible to generalize these modules so that even new binding constructs are easy to add to the base language.

3.1.4 Some Drawbacks

Unfortunately, the aggregate data approach suffers from some important drawbacks that detract from the benefits of modularity. Chief among these is the overhead of manipulating the aggregate data. Consider `mean-age`. The monolithic version of `mean-age` creates no aggregate data structures, but the list-based version of `mean-age` creates five intermediate lists during its execution. There is a time overhead associated with building these lists and taking them apart. Even more debilitating is the space overhead used to compute and store these lists. Under the standard Scheme evaluation model, memory must contain a list the size of the entire database whenever `map-age`, `map-one`, and `down-accumulate-sum` are applied. Furthermore, whenever the base case is reached in any of the computations specified in Figure 3.1, the size of the implicit control stack must be on the order of the database size as well. These problems plague any program representing cables by lists or trees.

The time overhead is very annoying but not devastating. It increases the running time of the monolithic `mean-age` by a constant multiplicative factor. In many cases it is reasonable to pay this price in order to reap the benefits of modularity. For example, because the modular version makes better use of the programmer's time (since it is easier to write, modify, and debug), it may actually be more cost-effective than a faster program that is harder to write.

The space overhead can be a more serious problem. Whereas the time requirements differed in a constant factor, the space requirements can differ in order of growth. The monolithic version of `mean-age` executes in constant space, while the list-based version requires both stack and heap space linear in the size of the database. A sufficiently large database can exhaust available memory in the list-based version, causing the program to fail. This is an unreasonable price to pay for modularity. Even though standardly available computer memories will continue to grow larger at a rapid pace, it is likely that the standard size of information chunks will grow at an even faster pace. The storage pitfalls of the aggregate data approach will become more problematic over time, not less so.

Furthermore, the straightforward aggregate data technique fails totally in cases where the aggregate structures are conceptually infinite. Infinite data structures can be a powerful way to modularize programs (see [ASS85]). An excellent example [Hug90] is decomposing a game program into a part that generates a game tree, and a part that examines the game tree. This supports modularity because the game tree generator can be designed as an independent unit without regard to the particular ways in which it will examined. But since game trees are typically infinite, this decomposition can stretch the above storage overhead problem beyond the capabilities of any finite memory.

Finally, although time and space overheads are the most important problems with the aggregate data approach, they aren't the only ones. Sometimes it is desirable to control the scheduling of operations from different slivers. For example, if two different slivers use I/O operations, we might want these interleaved in a certain fashion, or we might want all the operations in one sliver to happen before those in another. The aggregate data language is not powerful enough by itself to express such constraints. More generally, there are questions of how to combine the computational shapes of individual idioms to yield a desired shape for the composite. It is often possible to manually interweave several linear iterations into a single linear recursive computations, but the standard aggregate data approach does not allow us to talk in these terms.

3.1.5 Partial Solutions

There are a number of techniques for ameliorating the problems outlined above in special cases, but none is satisfactory in general.

`Cdr`-bashing

The `cdr`-bashing trick can be used to reduce the control space associated with list manipulations. Replacing every instance of `GLOM-LIST` with `CDR-BASHING-LIST-COLLECT` would remove the need for implicit stacks in all of the database examples. A similar trick, in conjunction with pointer reversal, could remove implicit stacks from the tree problems as well. But these tricks ultimately convert implicit stacks to explicit ones. They do not address the more fundemental problem of the space taken by the intermediate aggregate structures.

Lazy Data Structures

Lazy data structures are often suggested as a solution to the space problems described above, but this technique does not work in all cases.[1] A data structure is lazy when the computation of each of its components is delayed until it is required (if ever). In SPS programs based on aggregate data, lazy data can reduce space overhead and permit infinite data structures by effectively allowing a limited kind of coroutining between the slivers.

In some SPS programs, lazy data can eliminate the order-of-growth space overhead associated with the intermediate aggregate structures. For example, consider the following

[1] Lazy data structures are not to be confused with the more general strategy of *lazy evaluation*. Lazy evaluation introduces many space problems of its own (such as the *dragging tail problem* [Pey87]) that will not be detailed here.

procedure for counting the number of employees who earn more than a given amount:

```
(define (fat-cat-count threshold database)
  (running-sum
   (map-one
    (filter-salary threshold
                   (generate-records database)))))
```

Rather than manipulating lists, suppose that the subroutines manipulate Scheme *streams*, a form of lazy lists described in [ASS85]. A stream implementation of these subroutines appears in Figure 3.8.

```
(define (generate-records_stream database)
  (define (gen record)
    (if (end-of-database? record)
        the-empty-stream
        (cons-stream record (gen (next-record record)))))
  (gen (first-record database)))

(define (filter-salary_stream threshold records)
  (cond ((empty-stream? records) the-empty-stream)
        ((> (record-get (head records) 'salary) threshold)
         (cons-stream (head records)
                      (filter-salary_stream threshold (tail records))))
        (else (filter-salary_stream threshold (tail records)))))

(define (map-one_stream str)
  (if (empty-stream? str)
      the-empty-stream
      (cons-stream 1 (map-one_stream (tail str)))))

(define running-sum_stream
  (let ()
    (define (accum str sum)
      (if (empty-stream? str)
          sum
          (accum (tail str) (+ (head str) sum))))
    (lambda (stream) (accum stream 0))))
```

Figure 3.8: A stream-based implementation of the parts used by `fat-cat-count`.

The stream-based version of `fat-cat-count` executes in constant control and data space. This is a remarkable result: the space behavior of a monolithic version of `fat-cat-count` is achieved in a modular aggregate data program without any explicit concurrency or side effects. The operations of the subroutines are interleaved as a result of the delayed evaluation of the second argument of `cons-stream`. This interleaving enables the garbage collector to reclaim the intermediate storage allocated for each record by the time the next one is processed.[2] The side effects necessary for the `cdr`-bashing trick aren't needed here because they are effectively hidden by the memoization employed by streams to avoid the recomputation of delayed values [ASS85].

Unfortunately, lazy data is not a silver bullet. The subtle interactions between laziness and garbage collection can make it hard to predict the storage requirements of a program. As one example of the subtlety, compare the `running-sum`$_{stream}$ Figure 3.8 with the more straightforward version below:

```
(define (running-sum_stream stream)
  (define (accum str sum)
    (if (empty-stream? str)
        sum
        (accum (tail str) (+ (head str) sum))))
  (accum stream 0))
```

If the more straightforward version of `running-sum`$_{stream}$ is used in `fat-cat-count`, then constant space behavior cannot be guaranteed![3]

Even worse, lazy data doesn't always interact well with fan-out. Consider a stream implementation of `mean-age`, which exhibits fan-out by using the records generated from the database in two places:

```
(define (mean-age_stream database)
  (let ((records (generate-records_stream database)))
    (/ (running-sum_stream (map-age_stream records))
       (running-sum_stream (map-one_stream records)))))
```

Because Scheme evaluates procedure arguments in some sequential order, there is a point in the computation when one of the arguments to `/` has been completely evaluated but evaluation of the other has not yet begun. Due to the memoization of stream elements,

[2]The garbage collector may not actually reclaim the intermediate storage after every record is processed. But the fact that the storage can be reclaimed at any future point means that it not charged to the computation in space analysis.

[3]Here's why: Under the usual environment model of Scheme evaluation (see [ASS85]), the `stream` argument to `running-sum`$_{stream}$ is accessible from the environment in which `accum` is evaluated. Even though `stream` is never referenced within `accum`, its value can't be garbage collected until `accum` returns. But in the case of `fat-cat-count`, `stream` will be holding onto the stream created by `map-one`$_{stream}$, whose size is the number of fat cats in the database. So `fat-cat-count` is no longer constant space.

There are implementation strategies and alternate evaluation models under which `accum` won't unnecessarily hang onto `stream`. In these cases, `fat-cat-count` would still be constant space. However, the language doesn't *guarantee* this behavior.

memory must contain an intermediate data structure corresponding to the entire database at this point. If streams were not memoized, then there would be no storage overhead, but the database would have to be traversed twice. This latter case is equivalent to manually removing the fan-out from `mean-age`:

```
(define (mean-age_{no-fan-out} database)
  (/ (running-sum (map-age (generate-records database)))
     (running-sum (map-one (generate-records database)))))
```

In either case, lazy data does not help.

Hughes's Approach

Hughes improved the lazy data approach by supplying mechanisms that can guarantee desirable space behavior even for networks exhibiting fan-out [Hug83, Hug84]. He observed that in programs like `mean-age`, a constant space computation can only be achieved if the arguments to `/` are somehow computed together in lock step. That is, `generate-records` should not produce a new record until all the mappers and summers have completely processed the previous one; this way, no handle on the previous record needs to be stored.

Hughes showed that in functional languages with sequential argument evaluation it is *impossible* to express this kind of lock step evaluation in a modular way. To overcome the limitations of sequential argument evaluation, Hughes introduced mechanisms for parallel evaluation (`par`) and synchronization (`synch`). These are summarized in Figure 3.9.[4]

Using Hughes's `par` and `synch` constructs, the stream-based version of `mean-age` can be forced to exhibit constant space behavior as follows:

[4] Hughes originally posed his mechanism in a lazy functional language, so these versions have to be suitably modified to make sense in Scheme. Hughes's (`synch` E) actually returned a pair both of whose components held the value of E; but each component could only be accessed when *both* components had been requested.

- `(par` E`)` returns immediately with a placeholder for the result of the evaluation of E. The evaluation of E proceeeds concurrently with the rest of the computation. Any context requiring the actual value of the placeholder will wait until the value is available. `Par` is equivalent to the `future` construct provided by many Lisps [Hal85, Mil87, For91].
- `(synch` E`)` returns an object that suspends the computation of E. `Demand1` and `demand2` are procedures that demand such an object to return the value of its suspended computation, but the computation is only initiated when *both* `demand1` and `demand2` have been called on the object. Once computed, the value of E is memoized. `Synch` is similar to Scheme's `delay`, except that there are two different forcing functions, both of which must be invoked before the delayed expression is computed.

Figure 3.9: Scheme renditions of Hughes's `par` and `synch` constructs.

```
(define (mean-age_hughes database)
  (let ((records (synch-stream (generate-records_stream database))))
    (/ (par (running-sum_stream
              (map-age_stream
                (map-stream demand1 records))))
       (par (running-sum_stream
              (map-one_stream
                (map-stream demand2 records)))))))

(define (synch-stream str)
  (if (empty-stream? str)
      (synch the-empty-stream)
      (cons-stream (synch (head str))
                   (synch-stream (tail str)))))

(define (map-stream f str)
  (if (empty-stream? str)
      the-empty-stream
      (cons-stream (f (head str))
                   (map-stream (tail str)))))
```

`Mean-age`$_{hughes}$ uses `synch-stream` to associate a synchronization point with every database record and `par` to evaluate the operands of `/` concurrently. Since the age computation uses `demand1` to access a record and the length computation uses `demand2`, neither computation can race ahead of the other. It's as if both computations must pass through a series of locked doors, each of which has two locks, and each computation has they key to only one of the locks. As in the case of `fat-cat-count`, constant space behavior is once again guaranteed.

Hughes's technique is a good mechanism for controlling space behavior, but it has a few drawbacks:

- *Lack of abstraction*: The technique requires extending the sliver diagram for `mean-age` with two instances of `par` and three slivers that manage synchronization (`synch-stream` and two instances of `map-stream`). It would be preferable to somehow abstract over these parallelization and synchronization operations so that the original network structure could be maintained. Unfortunately, they are difficult to capture in a reusable fashion. For example, `synch-stream` can't simply be inserted into the definition of `generate-records`$_{stream}$ because other applications of this procedure might exhibit a fan-out other than two. Without extra support from the language, programmers are required to manage `par` and `synch-stream` explicitly — an error-prone prospect, especially since since `synch` raises the specter of deadlock.[5]

- *Weak synchronization*: Using `synch-stream` to replace every instance of fan-out in a sliver diagram does not force all the slivers in the diagram to compute in lock step. Since synchronization is local, not global, there may be a lag between two sliver computations that's related to the number of `synch-stream`s that appear between them. This is usually not very important, but can be troublesome if tighter operator interleaving is desired.

The concurrency and synchronization mechanisms underlying my technique are closely related to Hughes's `par` and `synch` except that they address the two drawbacks above. In particular, my technique makes it possible to hide the mechanisms inside of sliver abstractions so that the programmer does not have to deal with them.

Program Transformations and Compilation Techniques

There are a large number of program transformations and compilation techniques that can eliminate some intermediate data structures from aggregate data programs by fusing connected slivers. They are all essentially high-level versions of the classic low-level *loop*

[5] *Deadlock* is a state in which a computation can make no progress. Deadlock can arise in the presence of `synch` if `demand2` cannot be applied until the successful return of `demand1`, or vice versa.

fusion technique performed by many optimizing compilers [ASU86]. As a typical example of these techniques, the Scheme expression

```
(map f (map g l))
```

could be replaced by

```
(map (lambda (x) (f (g x))) l).
```

The later expression is preferable because no list is constructed for the ouput of `g`.

The problem with existing transformation and compilation techniques is that they either provide no guarantees or they only work on a limited class of programs. Algebraic transformation techniques [DR76, Dar82, Bac78, Bel86, Bir89a, Bir86, Bir88] perform transformations like the above `map` removal, but systems that automatically apply these transformations do not guarantee that all intermediate data will be removed. Because programmers cannot depend on the transformations, they must seek other methods of controlling the space behavior of their programs. APL compilation techniques [GW78, Bud88] suffer from the same problem.

The listlessness [Wad84, Wad85] and deforestation [Wad88, Chi92, GLJ93] techniques pioneered by Wadler *do* provide guarantees, but they are rather limited in applicability. Listlessness handles a subclass of list programs, but no trees. Deforestation can eliminate both lists and trees, but only in networks that exhibit no fan-out.

The most impressive of the transformation approaches is Waters's *series* technique [Wat91, Wat90]. Series is a linear datatype that corresponds to a sequence of the values assumed by a state variable during an iteration. The series compiler can transform a large class of series networks, including those with fan-in, fan-out, and filtering, into efficient loops. The most important aspect of series is that there is an explicit set of easy-to-check restrictions that the programmer can verify to determine whether a given network can be efficiently compiled. The compiler generates a warning when these restrictions are violated.

Alas, series is limited to the expression of linear iterative computations. It cannot handle tree-shaped computations, or even linear recursions. Cyclic data dependencies are not allowed, even though these are sometimes useful for program decomposition.[6] Finally, because it is based on static analysis, series requires that the network of series operators be determinable at compile-time. This limits expressiveness by outlawing the dynamic configuration of the network at run-time. The lock step processing technique developed in Chapter 5 is the basis for dynamic version of series that addresses all of these issues.

The problem with all of the transformations described in this section is that they are too restrictive (e.g., can't handle tree-shaped data, fan-out, or recursion) and/or they fail to provide provide the programmer with sufficient guarantees about improvements. The upshot is that programmers often shun the aggregate data approach and instead embrace the fine-grained control of the monolithic style.

The storage overhead problem is a classic example of how modularity can preclude a programmer from expressing desired behavior. The aggregate data approach hides the operational details of the slivers inside of procedures specified wholly in terms of their input/output behavior. The programmer has no hook into how the procedures work, and therefore cannot express details like lock step evaluation.

3.2 The Channel Approach

In the channel approach, SPS networks are viewed as interconnected processes that communicate via data channel defined by the cables. Whereas the focus in the aggregate data approach is the data transmitted on a cable, the focus in the channel approach is the process that sends and receives data from the channel. A characteristic of the channel approach is that the collection of elements transmitted across a channel is not treated as a single entity. In the channel approach, processes are usually assumed to be independent threads of control. They may be executing concurrently, or they may be coroutining in some fashion.

The channel approach is supported by numerous languages and systems. Hoare's CSP is the canonical version of this approach [Hoa85]; Unix pipes [KP84] is one of the most widely used. Other examples of the channel approach include: communicating threads

[6] Waters argues that such cycles make programs harder to understand, and therefore should be avoided at all costs.

[Bir89b, CM90], producer/consumer models (CLU iterators [L^+79], Id's I-structures and M-structures [ANP89, Bar92], Linda [CG89]) and dataflow ([Den75], [WA85], [DK82]).

Below, we explore both coroutining and concurrent versions of the channel approach. We will see that the channel approach can provide reasonable control over space behavior but that it is not a very good approach for expressing linear recursions and tree shaped computations.

3.2.1 Coroutining Example

As a demonstration of coroutining, consider a simple organization in which each sliver is represented as an state-maintaining object that keeps track of the source object for each of its input cables (this is the representation of the "channel"). Computation will be performed in a demand-driven fashion. If an object receives a request for the next value on one of its output cables, it may in turn request values from its input cables in order to satisfy the request.[7]

Figures 3.10 and 3.11 present encodings of the database idioms under this organization. Every object (except for record generators) has a `source` variable that indicates the source of its input wire. Each object is represented as a thunk (parameterless procedure) that returns its next output every time it is called. The `demand!` procedure enhances the readability of this usage pattern:

```
(define (demand! object) (object))
```

When an object runs out of values to produce, it returns a distinguished `done` value:[8]

```
(define done '(*done*))
(define (done? obj) (eq? obj done))
```

A detail: the `set!`s used within the accumulators guarantee that they will return `done` after producing the accumulated value.

Here's a version of the `fat-cats` program in this approach:

[7] In more complex organizations, objects might also keep track of the targets of their output cables, and computation might exhibit both data-driven and demand-driven aspects.

[8] An alternate approach is for every object to handle both an "are you done?" message and a "give me your next value" message. It turns out that this alternate approach unduly complicates the filtering idiom in this style (try it and see!).

```
(define (make-record-generator_cor database)
  (let ((record (first-record database)))
    (lambda ()
      (if (end-of-database? record)
          done
          (let ((val record))
            (set! record (next-record record))
            val)))))

(define (make-running-summer_cor source)
  (define (accum sum)
    (let ((next (demand! source)))
      (if (done? next)
          (begin (set! accum (lambda (x) done))
                 sum)
          (accum (+ next sum)))))
  (lambda () (accum 0)))

(define (make-stacking-list-collector_cor source)
  (define (accum)
    (let ((next (demand! source)))
      (if (done? next)
          (begin (set! accum (lambda () done))
                 '())
          (cons next (accum)))))
  (lambda () (accum)))
```

Figure 3.10: Generating and accumulating idioms for the database example in the coroutining technique.

```
(define (make-mapper-maker_cor fun)
  (lambda (source)
    (lambda ()
      (let ((next (demand! source)))
        (if (done? next)
            done
            (fun next))))))

(define make-age-mapper_cor
  (make-mapper-maker_cor (lambda (r) (record-get r 'age))))

(define make-name-mapper_cor
  (make-mapper-maker_cor (lambda (r) (record-get r 'name))))

(define make-1-mapper_cor
  (make-mapper-maker_cor (lambda (r) 1)))

(define (make-salary-filter_cor threshold source)
  (define (next)
    (let ((record (demand! source)))
      (cond ((done? record) done)
            ((> (record-get record 'salary)
                threshold)
             record)
            (else (next)))))
  (lambda () (next)))
```

Figure 3.11: Filtering and mapping idioms fort the database example in the coroutining technique.

```
(define (fat-cats_cor threshold database)
  (demand! (make-stacking-list-collector_cor
            (make-name-mapper_cor
             (make-salary-filter_cor threshold
                                     (make-record-generator_cor database))))))
```

This resembles the aggregate data approach in the way that the interconnecton of slivers is specified by nesting expressions. However, no intermediate aggregate structures are created. Instead, demand and values pass back and forth between the objects in a coroutining fashion. This computational pattern closely resembles the evaluation of a lazy data version of `fat-cats`.

The `mean-age` program is harder to express in this approach due to fan-out. The naive approach, shown below, fails to work as desired:

```
;;; Doesn't work because of fan-out!
(define (mean-age_cor-wrong database)
  (let ((generator (make-record-generator_cor database)))
    (/ (demand! (make-running-summmer_cor
                 (make-age-mapper/dist generator)))
       (demand! (make-running-summmer_cor
                 (make-1-mapper/dist generator))))))
```

Because requesting a record from `generator` changes its state, the two mappers connected to `generator` receive different record sequences!

To avoid this problem without traversing the database twice, it is necessary to buffer values produced by the generator until both mappers have consumed them. This can be done in general by the using the `copy`$_{cor}$ routine shown in Figure 3.12 to represent cable fan-out in a sliver diagram. `Copy`$_{cor}$ uses a buffer to effectively return two copies of a single object. Using this routine, `mean-age` can be correctly implemented as:

```
(define (mean-age_cor database)
  (mlet (((gen1 gen2) (copy_cor (make-record-generator/dist database))))
    (/ (demand! (make-running-summer_cor
                 (make-age-mapper_cor gen1)))
       (demand! (make-running-summer_cor
                 (make-1-mapper_cor gen2))))))
```

Introducing buffering makes it possible to handle fan-out, but it brings with it the same space overhead problems that hamper the aggregate data approach. In a language with sequential argument evaluation, the `mean-age`$_{cor}$ procedure will end up buffering the entire database at some point. As in the aggregate data approach, the problem with the

```
(define (copycor source)
  (let ((queue (make-queue))
        (slowpoke 0)) ; 0 = neither copy behind;
                      ; 1 = first copy behind;
                      ; 2 = second copy behind.
    (define (make-copy my-id other-id)
      (lambda ()
        (if (= my-id slowpoke)
            ;; If I'm behind, get next value from queue.
            (let ((head (dequeue! queue)))
              (if (queue-empty? queue)
                  (set! slowpoke 0))
              head)
            ;; If I'm not behind, get next value from source and queue it.
            (let ((next (demand! source)))
              (begin
                (enqueue! next queue)
                (set! slowpoke other-id)
                next)))))
    ;; Return a list of two thunks that act like SOURCE.
    (list (make-copy 1 2) (make-copy 2 1))))
```

Figure 3.12: A routine that returns two copies of a given object. The procedures make-queue, enqueue!, dequeue!, and queue-empty? implement a queue abstraction that is not detailed here.

coroutining technique is that the modules preclude the programmer from expressing fine-grained control. In both cases, some form of concurrency and synchronization are required to express desired control.

A new problem exhibited by coroutining is that tree-shaped computations are hard to handle. Using demands to request the next value from an input cable is fine when the values are arranged linearly in sequences, but is problematic when they are arranged as trees. How would `alpha-rename` be coded in the coroutining technique? There are several approaches, none of which is entirely satisfactory:

- Encode the tree as a linear sequence of its contents along with extra information indicating the shape of the tree. This kind of encoding is used all the time to transmit structured information over physical wires. Yet, a programmer doesn't want to have to think in these terms; at the very least, there need to be abstractions for the encoding and decoding process.

- Associate with each request an address that indicates the desired element. This complicates both requestor and requestee by requiring them to keep track of address information.

- Have each object respond to `demand!` by returning subtree objects in addition to the usual value. The requester then can explicitly choose which subobject to `demand!` next. This complicates each requester by forcing it to manage the subobjects.

3.2.2 Concurrent Process Example

We address the concurrency problem of the coroutining technique by considering a model in which the sliver processes are concurrent. The processes need not actually be running in parallel on separate physical processors; for our purposes, simulated concurrency on a single processor is perfectly adequate.

The Database Example

Versions of `mean-age` and `fat-cats` written in the concurrent process technique are shown in Figure 3.13. They make use of the idioms presented in Figures 3.14 and 3.15. In this code, we assume that Scheme has been extended with some constructs that support concurrency:[9]

- (`cobegin` *exp1 exp2* ...) evaluates each of the subexpressions in its own concurrent thread. Evaluation of the subexpressions may be arbitrarily interleaved. `Cobegin` returns only when all component thunks have returned; its return value is a list of the subexpression values.

- (`make-channel`) returns a new channel object, which acts as a FIFO communication queue between concurrently executing threads.

- (`send!` *channel value*) tacks *value* on the end of the *channel* queue and returns *value*.

- (`receive!` *channel*) removes the first value from the *channel* queue and returns it. If the queue is empty, then `receive!` blocks until a value is available to read.

Channels are assumed to be buffered communication queues in which sends and receives are performed by side effect, but there are many other communication models that could have been chosen. As in the coroutining technique, `done` and `done?` are used to indicate the termination of a value sequence.

In `mean-age`$_{conc}$ and `fat-cats`$_{conc}$, each cable is represented explicitly as a channel object, and each sliver is represented as a channel-manipulation procedure. The `cobegin` allows the slivers conceptually to execute in parallel, though on a sequential machine their computations are actually interleaved. Most slivers wait for values from their input channels, and then produce values on their output channels. The blocking behavior of `receive!` on an empty channel is a synchronization mechanism that forces slivers to wait for values to become available on their input channels before proceeding. This results in a data-driven model of evaluation that stands in contrast with the demand-driven evaluation of the lazy data technique and the coroutining technique.

The rationale behind the `copy`$_{conc}$ is the same as that for `copy`$_{cor}$ in the previous example. Since `receive!` works by side effect, two consumers sharing the same channel will not see the same sequence of values. Copying the values to two separate channels decouples the

[9]Except for a few cosmetic changes changes, these constructs are the ones described in [JG89].

```
(define (mean-age_conc database)
  (let ((gen->copy      (make-channel))
        (copy->map-age  (make-channel))
        (copy->map-one    (make-channel))
        (map-age->sum1  (make-channel))
        (map-one->sum2    (make-channel)))
    (mlet (((_ _ _ _ sum1 sum2)
            (cobegin
              (generate-records_conc database gen->copy)
              (copy_conc gen->copy copy->map-age copy->map-one)
              (map-age_conc copy->map-age map-age->sum1)
              (map-one_conc copy->map-one map-one->sum2)
              (running-sum_conc map-age->sum1)
              (running-sum_conc map-one->sum2))))
      (/ sum1 sum2))))

(define (fat-cats_conc threshold database)
  (let ((gen->filter    (make-channel))
        (filter->map    (make-channel))
        (map->list      (make-channel)))
    (mlet (((_ _ _ name-list)
            (cobegin
              (generate-records_conc database gen->filter)
              (filter-salary_conc threshold gen->filter filter->map)
              (map-name_conc filter->map map->list)
              (stacking-list-collect_conc map->list))))
      name-list)))
```

Figure 3.13: Versions of `mean-age` and `fat-cats` in the concurrent process technique. The `mlet` construct is the pattern matching version of `let` introduced earlier; the underscore character '_' is used in patterns to indicate unnamed slots.

```
(define (generate-records_conc database out)
  (define (gen record)
    (if (end-of-database? record)
        (send! out done)
        (begin
          (send! out record)
          (gen (next-record record)))))
  (gen (first-record database)))

(define (running-sum_conc in)
  (define (sum ans)
    (let ((next (receive! in)))
      (if (done? next)
          ans
          (sum (+ next ans)))))
  (sum 0))

(define (stacking-list-collect_conc in)
  (define (accum)
    (let ((next (receive! in)))
      (if (done? next)
          '()
          (cons next (accum)))))
  (accum))
```

Figure 3.14: Generating and accumulating idioms for the database example in the concurrent process approach.

```
(define (make-map_conc f)
  (define (map in out)
    (define (loop)
      (let ((next (receive! in)))
        (if (done? next)
            (send! out done)
            (begin
              (send! out (f next))
              (loop)))))
    (loop))
  map)

(define map-age_conc  (make-map_conc (lambda (r) (record-get r 'age))))
(define map-name_conc (make-map_conc (lambda (r) (record-get r 'name))))
(define map-one_conc  (make-map_conc (lambda (x) 1)))

(define (filter-salary_conc threshold in out)
  (define (loop)
    (let ((next (receive! in)))
      (if (done? next)
          (send! out done)
          (begin
            (if (> (record-get next 'salary)
                   threshold)
                (send! out next))
            (loop)))))
  (loop))

(define (copy_conc in out1 out2)
  (define (loop)
    (let ((next (receive! in)))
      (begin
        (send! out1 next)
        (send! out2 next)
        (if (not (done? next)) (loop)))))
  (loop))
```

Figure 3.15: Mapping, filtering, and copying idioms for the database example in the concurrent process approach.

consumers from each other. An alternate approach would be to share a single channel but provide each consumer with its own pointer into the channel queue.

Storage Overhead

The concurrent process technique offers the possibility of better storage behavior than the other approaches, but does not guarantee it. Consider `mean-age`$_{conc}$. If all of the component processes are operating in lock step, then no channel queue ever contains more than one element and the program as a whole uses only constant space. However, at the other extreme, the `map-age`$_{conc}$ process might send all database ages to its output channel before the `running-sum`$_{conc}$ at the other end of the channel receives the first one. In this case, the `map-age->sum1` channel requires storage linear in the size of the database.

Channel storage requirements can be reduced by constraining the unbounded nature of channel queues. The problem with unbounded channels is that they allow a producer to race arbitrarily far ahead of one of its consumers. A common solution is to limit the number of values that the channel can buffer.

In the most restrictive approach, channels aren't allowed to buffer any values; instead, the processes at the ends of the channel engage in a *rendezvous* in which a `send!` does not return in the sending process until a `receive!` has been executed in the receiving process. This is the approach adopted by CSP [Hoa85]. A rendezvous-based version of `mean-age`$_{conc}$ would be guaranteed to require only constant space.

A less restrictive approach is a *bounded queue*, in which there is an upper limit on the size of a queue; when the queue reaches this limit, a `send!` blocks until the queue gets smaller. `mean-age`$_{conc}$ would also use only constant space under this approach, though the constant would be larger than in the rendezvous case. The bounded queue approach is a standard means of implementing flow control between independent processes [Bir89b].

Why not simply adopt a policy in which all the channel queues are bounded? The problem with this is that there are problems that require unbounded queues. Consider an `append`$_{conc}$ process with two input channels and one output channel that first copies all elements from the first input channel to the output channel and then copies all elements from the second input channel to the output channel (Figure 3.16).

```
(define (append_conc in1 in2 out)
  (define (copy1)
    (let ((next (receive! in1)))
      (if (done? next)
          (copy2)
          (begin (send! out next)
                 (copy1)))))
  (define (copy2)
    (let ((next (receive! in1)))
      (if (done? next)
          (send! out next)
          (begin (send! out next)
                 (copy2)))))
  (copy1))
```

Figure 3.16: A procedure that sends to its output channel the result of appends the values from its two input channels.

If $\texttt{append}_{conc}$ is used in the following `twice` procedure,

```
(define (twice in out)
  (mlet (((c1 c2) (copy_conc in)))
    (append_conc c1 c2 out)))
```

then the second input channel `c2` to $\texttt{append}_{conc}$ requires a queue whose size is the number of elements in the `in` channel. Since this size may be arbitrarily large, bounded queues do not suffice in general.

A reasonable compromise is to let the programmer choose between bounded and unbounded channels. In addition to the unbounded `(make-channel)`, the language could also provide a `(make-bounded-channel` *num*`)` constructor that creates a channel whose queue length is bounded by *num*. Then for cases like `mean-age`, the programmer has a means of expressing the constant-space nature of the computation while maintaining the modular SPS structure. Its ability to support both modular decomposition and some means of constraining control distinguishes the concurrent process technique from the other techniques we have studied.

Deadlock

While the concurrent process technique has some nice properties, there are a number of issues that detract from it. Foremost among these is that concurrent programs can sometimes enter a wedged state called *deadlock* in which processes aren't done executing, but none can make any progress. Here is a simple example of a program that deadlocks:

```
(let ((c1 (make-channel))
      (c2 (make-channel)))
  (cobegin
    (begin (receive! c2) (send c1 19))
    (begin (receive! c1) (send c2 23))))
```

Each of the two processes created by the `cobegin` attempts to receive a value from the other process before it sends a value to the other process. Since neither process can make headway, the program is stuck and no value is returned.

The problem with the above example is that the sequential ordering constraints implied by the `begin`s are too strong and cannot be satisfied. Deadlock can often be avoided by removing spurious ordering constraints. For instance, the above example can be made to work by changing the `begin`s to `cobegin`s:

```
(let ((c1 (make-channel))
      (c2 (make-channel)))
  (cobegin
    (mlet (((_ ans1)
            (cobegin (receive! c2) (send! c1 19))))
      ans1)
    (mlet (((_ ans2)
            (cobegin (receive! c1) (send! c2 23))))
      ans2)))
```
$\overset{\text{eval}}{\Longrightarrow}$ `(19 23)`

This change is reasonable because the `send!`s do not depend on the values of the `receive!`s. However, when there is an inherent dependency loop, deadlock is unavoidable and indicates a program bug. Here is a deadlocking expression containing such a loop:

```
(let ((c1 (make-channel))
      (c2 (make-channel)))
  (cobegin
    (send c1 (receive! c2))
    (send c2 (receive! c1))))
```

The above examples are contrived, but spurious deadlocks are easy to come by in practice. Avoiding them requires a defensive programming style that relaxes unnecessary ordering constraints. For instance, the unnecessary ordering of `send!`s in the `copy`$_{conc}$ procedure from Figure 3.15 can insidiously lead to deadlock in some contexts. This procedure can be written more robustly by decoupling the `send!`s with a `cobegin`:

```
(define (copy_conc-better in out1 out2)
  (define (loop)
    (let ((next (receive! in)))
      (begin
        (cobegin               ; Relax ordering of SEND!s
          (send! out1 next)
          (send! out2 next))
        (if (not (done? next)) (loop)))))
  (loop))
```

In the presence of bounded channels, deadlock is even more of a threat because it can be caused simply by choosing a bound that is lower than actually required.

Deadlock complicates the concurrent processing approach to SPS by requiring the programmer to reason more carefully about sliver composition than in other styles. In the other approaches, the meaning of a network is easy to determine from the meaning of the parts; as long as there are no *directed* loops between slivers, there are no surprises. However, in the concurrent processing style, deadlock can arise from *undirected* loops[10] or channels whose buffers aren't big enough. The extra reasoning necessary to show that a program does not have deadlock is part of the price of additional control.

Other Issues

Deadlock is not the only issue that complicates SPS programs written in the concurrent processing style. Here are some others:

- *Synchronization:* Deadlock arises from overconstrained control. But underconstrained control also leads to problems, usually involving side effects. Consider the following version of copy:

```
(define (copy_conc-worse in out1 out2)
  (define (loop)
    (let ((next (receive! in)))
      (cobegin
        (send! out1 next)
        (send! out2 next)
        (if (not (done? next)) (loop))))) ; ***
  (loop))
```

[10] An undirected loop between slivers is formed by any two distinct paths from one to another.

This differs from $\texttt{copy}_{conc-better}$ in that the starred line appears inside the `cobegin`. This allows the next value to be received in parallel with sending the current value. Unfortunately, it also allows the next value to be sent *before* the current value is sent. This changes the input/ouput behavior of `copy` in a major way.

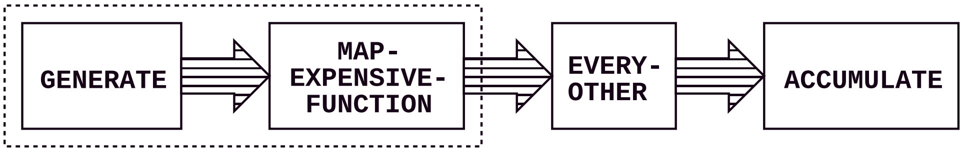
Underconstrained control is a classic problem whenever concurrent processes communicate via shared mutable data. The solution is to introduce extra constraints in the form of synchronization. Typical synchronization mechanisms include locks [Bir89b], semaphores [Dij68], monitors [Hoa74], I-structures [ANP89], M-structures [Bar92], and Hughes's `synch` construct [Hug83, Hug84].

- *Termination:* A `cobegin` does not return until all its subprocesses have returned; this kind of behavior is typical of fork/join parallelism. But getting all the subprocesses to terminate can be tricky. Imagine an accumulator that only consumes the first few values of a long or unbounded sequence produced by a generator. The accumulator returns quickly; but what forces the generator to stop and return? For these kinds of situations, the concurrent process technique often requires a special means of terminating processes.

  An alternate approach is to allow a process to return before it terminates. This decoupling is employeed in so-called *eager* or *lenient* evaluation strategies ([Tra88, Hal85, Mil87]) in which computations may continue to execute after a final answer has been returned. Some means of garbage-collecting these superfluous computations is necessary to avoid wasting resources.

- *Tree-shaped Data:* Like the coroutining technique, the concurrent process technique has difficulty expressing tree-shaped computations. The same circumventions suggested for coroutining work here as well. But a more satisfying approach is to mix aspects of the data aggregate approach and the concurrent process approach. For example, rather than transmitting a tree over a single channel, why not transmit a tree over a tree of channels, where each channel transmits exactly one element? This is the approach taken by languages, such as Id [AN89] and Linda [CG89], that exhibit *producer/consumer parallelism.* These languages support *non-strict data structures* that can be manipulated by a program before their components have been computed. Any attempt to reference an uncomputed component results in a computation that is blocked until the component is actually there. Thus, components of a non-strict data structure are essentially channels that can communicate only a single element before being "used up".

- *Unnecessary Computation:* Data-driven evaluation can often perform unnecessary operations. As a simple example, imagine a network where an expensive mapper precedes a filter that passes only every other element:



  In this case, much of the work being done by the mapper is for naught. This contrasts with demand-driven evaluation, in which only those operations necessary for the final result are performed. An obvious fix is to swap the components, but the mapper may be packaged together with other slivers in a way that makes this infeasible.

  In systems that provide real parallelism (i.e., many physical processors), it is often possible to reduce overall computing resources by wasting or repeating some computation. (The data parallel techniques discussed in [HS86a] are an excellent example of this phenomenon.) In such systems, demand-driven techniques can reduce performance rather than improving it. However, in systems providing only simulated parallelism (i.e., a single processor), demand-driven techniques may still be advantageous. Pingali and Arvind show how demand-driven computation can be simulated within a dataflow model [PA85], [PA86].

- *Overconstrained Operation Order:* The linearity of channels sometimes overconstrains the order in which operations are applied. Consider the mapping routines in Figure 3.15. These routines force the mapped function to be applied to each record in the order in which it appears in the database. Compare this behavior with the mapping functions for the aggregate data approach. In the latter case, the mapped function can happen in any order with respect to the recursive mapping. This unspecified order can increase the modularity of components like mappers, which can then be used in contexts that constrain the order in different ways. It is unwise to specify operation scheduling too early!

3.3 Other Techniques

Here we discuss a few common modularity techniques that are clearly relevant to the expression of loops and recursions, but can't easily be characterized within the aggregate data approach or the channel approach.

3.3.1 Higher Order Procedures

In a programming language that supports higher-order procedures, it is natural to capture general loops and recursions as procedures. For example, here are Scheme procedures encapsulating iteration and linear recursion:

```
(define (iterate initial-state done? down final)
  (define (iter state)
    (if (done? state)
        (final state)
        (iter (down state))))
  (iter initial-state))

(define (recur initial-state done? base down up final)
  (define (rec state)
    (if (done? state)
        (base state)
        (up state (rec (down state)))))
  (final (rec initial-state))))
```

While this is an elegant and powerful strategy for capturing control constructs, it does not provide the kind modularity supported by the signal processing style.[11] For example, implementing `mean-age` via `iterate` requires the programmer to manually interweave the `done?` parts of the component slivers, the `down` parts of the components, and the `final` parts of the components. This is hardly an improvement over the monolithic approach! The

[11] Another problem is that general tree walkers are considerably more complex than the simple linear examples shown here.

problem is that the looping control structure is still centralized in `loop` rather than being distributed over components.

Modularity could be achieved in this approach by representing an iterative sliver as a record of procedures that described the sliver's contribution to each of the arguments of the abstracted control structure. For example, an iterative sliver would be a record of `initial-state`, `done?`, `down`, and `final` components. However, it would also be necessary to define an appropriate means of combination for such records, which would be complicated in general. While this approach may not be reasonable for programmers, it can be good idea for compilers. Indeed, the notion of gluing together corresponding fragments of different slivers is at the heart of Waters's series compiler [Wat91].

3.3.2 Looping Macros

Many versions of Lisp have supported complex macros that capture certain looping idioms.[12] For example, in Common Lisp [Ste90], the `mean-age` procedure can be written as follows:

```
(defun mean-age (database)
  (loop for record = (first-record database)
                   then (first-record (next-record record))
        until (end-of-database? record)
        sum (record-get record 'age) into total
        count record into length
        finally (return (/ total length))))
```

Here, `for`, `until`, `sum`, `count`, and `finally` all introduce clauses into the occurence of the `loop` looping macro. Each clause is intended to express a common iterative processing idiom in a convenient syntax.

As with the higher-order procedure approach, looping macros suffer modularity problems. Again, there is a centralized control structure that is not distributed over the components. There is no way to abstract over a collection of clauses to reuse them in other occurrences of `loop`. While some clauses concisely capture an idiom (e.g., the `sum` and `count` clauses), other idioms are spread over several clauses (e.g., record generation). This approach is essentially an alternate syntax for writing a monolithic iteration.

[12] As far as I know, there are not any similar macros that capture the idioms of tree recursions. Designing such a macro would be an interesting project.

3.3.3 Attribute Grammars

Attribute grammars are a formalism invented by Knuth [Knu68] for declaratively specifying the decoration of tree nodes with named attributes. Though originally intended for describing the semantics of programming languages based on their grammars, today they are mainly used for syntax-directed compilation techniques (see [DJL88] for an overview).

Attribute grammars are related to slivers because (1) they are a language for describing a class of tree computations (2) they support a crude notion of shape. Attribute grammars specify computations on a tree (typically a parse tree) by indicating how an attribute at one node in a tree is calculated from attributes at the current node, its parent node, or its children nodes. Attributes are classified according to dependency information: an attribute computed from parent nodes is said to be *inherited*, while one computed from children nodes is said to be *synthesized*. "Inherited" and "synthesized" roughly correspond to *parallel down* and *parallel up* computation shapes. Indeed, it is possible to view attribute grammars as defining recursive functions [Joh87] or procedures [Kat84].

Classical attribute grammars suffer from a lack of modularity because they distribute the specification of attribute computations across all the different types of tree nodes. However, in recent years, there has been a flurry of activity on *modular attribute grammars*, which strive to group all the computations of a given attribute into a modular unit [DC90, Ada91, FMY92, KW92, Wat92].

While attribute grammars can be a powerful declarative framework for specifying tree computations, they do not seem to be good languages for controlling the behavior of these computations:

- As far as I can tell, most of the attribute grammar frameworks assume that the entire tree being decorated resides in memory. This is an exponential difference in space complexity for monolithic tree-recursive procedures that require space proportional only to the depth of the tree.

- Attribute grammars don't supply a means of expressing tail recursion. To return a result from the leaf of a tree, it is necessary to pass it through all the intermediate nodes back to the root, even if there are no pending computations to be performed on it.

- The declarative nature of attribute grammars can make it difficult to predict how many passes the attribute computation will make over the tree. (There are some frameworks, such as one-pass attribute grammars [Kos84], that do limit the number of passes).

- Attribute grammar formalisms are usually not designed to express general tree computations; they are typically tied to parsing technology and are used mainly to specify language implementations.

- Modular attribute grammar formalisms typically involve ad hoc mechanisms for specifying component connectivity. Connections are much more natural to express in the aggregate data and channel approaches.

3.4 Summary

We have examined in detail issues of modularity and control for a linear and tree-based example. The following themes stand out:

- Modularity arguments suggest that some programs can be structured like signal processing networks of mix and match slivers that embody idiomatic loops and recursions. This approach is called the signal processing style of programming.

- The modularity of slivers is enhanced when their specification leaves open certain details that can be filled in by the context in which they are used. Examples of this theme include:

  - *Higher-order procedures* capture general patterns while allowing parameters to be supplied where they are used.

  - *Lazy data* permits the specification of potentially infinite data structures, but these are only computed to the extent required by the context in which they're used.

  - *Concurrency* allows operations in different slivers to be interleaved, subject to the ordering constraints that result from their connection. The reusability of concurrent processes is increased when spurious ordering constraints on their computations are removed.
  - *Unspecified sequential evaluation of arguments* gives a limited form of operator interleaving.

- *Side effects* are a crucial mechanism both for achieving modularity and specifying fine-grained control. They aid modularity by localizing computation and reducing the interfaces between parts. They permit fine-grained control of space resources and operation order, as in the `cdr`-bashing list collection example.

- Traditional forms of implementing input/output modularity often preclude the programmer from controlling operational details of a program at a fine-grained level. Such details include time and space complexity, operation order, and limiting unnecessary computations.

- Standard techniques for SPS programming have benefits and drawbacks along various dimensions:

  - *Storage requirements* and operation scheduling are difficult to control in the aggregate data and coroutining techniques. They are easier to control in presence of concurrency, but then it is necessary to contend with issues such as deadlock and termination.
  - *Fan-out* in the cables of sliver diagrams causes no problems for the aggregate data style, but requires special handling in the channel techniques. Undirected cycles (formed by fan-out in combination with fan-in) can also lead to deadlock in the concurrent process technique.
  - *Unnecessary computation* can be avoided by using demand-driven evaluation. This form of evaluation is natural in the lazy aggregate data style and the demand-driven coroutining approach. However, it is in conflict with the natural data-driven evaluation of concurrent programs.

– *Tree-shaped data* is easy to handle in the aggregate data approach, but is problematic in the channel approaches.